UCRL-CONF-209786

PyHelp - An automatic multi-output documentation generator for Python

W. I. Nissen

February 17, 2005

NUCLEAR EXPLOSIVES CODE DEVELOPERS CONFERENCE 2004
LIVERMORE, CA, United States
October 4, 2004 through October 7, 2004

PyHelp - An automatic multi-output documentation generator for Python (U)

W. I. Nissen

Lawrence Livermore National Laboratory, Livermore, California, 94551

The DRACO code creates geometry and meshes through a command-line Python interface consisting of hundreds of classes and modules which must be accompanied by current documentation. The standard Python utility pydoc *performs introspection on objects and prints their associated documentation strings verbatim. However,* pydoc *supports only very rudimentary formatting and cannot produce printable documentation. We decided to modify* pydoc *to process formatted "docstrings" and use the Doxygen tool to generate the needed forms of documentation. (U)*

Introduction

Codes with complex interfaces often require substantial effort to keep user documentation current with interface changes. The DRACO code creates geometry and meshes through a command-line Python interface consisting of hundreds of classes and thousands of functions. A previous attempt to write documentation manually quickly fell out of date, so the development team needed to find an alternative. The existing tools did not provide the flexibility we needed, and the team was already conversant in Doxygen, a C++ code-documenting utility with a simple tag-based markup. Python comes with a utility, *pydoc*, that performs introspection on objects and prints their docstrings verbatim. However, *pydoc* supports only very rudimentary formatting and cannot produce printable documentation. Thus we decided to create "docstrings" written in Doxygen syntax for each object and process them with a modified *pydoc* to generate the needed forms of documentation.

Behavior

The primary requirement for PyHelp was making documentation easy to write. Rather than explain the detailed syntax, an annotated example of a documented function (Fig. 1) serves to demonstrate most of the features. In general, all markup is done with "tags" that start with a backslash '\' and may or may not be followed by arguments.

```
def composite( self, edgeList=None, _showWarning = True ) :
```

The definition of the function, part of the regular source code. The name and arguments are taken from here.

```
    """Create a composite edge.
```

A one-line description, to be displayed when asking for brief help.

```
    \call instance.composite( )
    \call instance.composite( edgeList )
```

Legal calling sequences for the function.

```
    \param edgeList edge or list of edges
```

The arguments of the function, one per \param.

```
    \return None
```

The return value of the function.

```
    \description
    Creates a composite edge by joining two or more consecutive edges.
    Edges must be part of a sheet, solid, or wire body, and must be
    given in the order they are connected end-to-end. Vertices joining
    the edges must have exactly two edges. See also \ref
    Face.Face.composite.
```

The description of the function. Note the use of a link to another object.

```
    \example
    wire=Wire.FromPoints([(0,0,0),(1,0,0),(2,0,0),(3,0,0)])
    e0=Entity.Edge(0)
    e1=Entity.Edge(1)
    e0.composite(e1)
    \endexample
```

An example demonstrating the use of the function.

```
    Auto-compositing is especially useful when a CAD part with
    many edges has been used to create the geometry, because it
    only requires that you specify one edge.
    """
    pass
```

The function itself would be defined here, but is omitted.

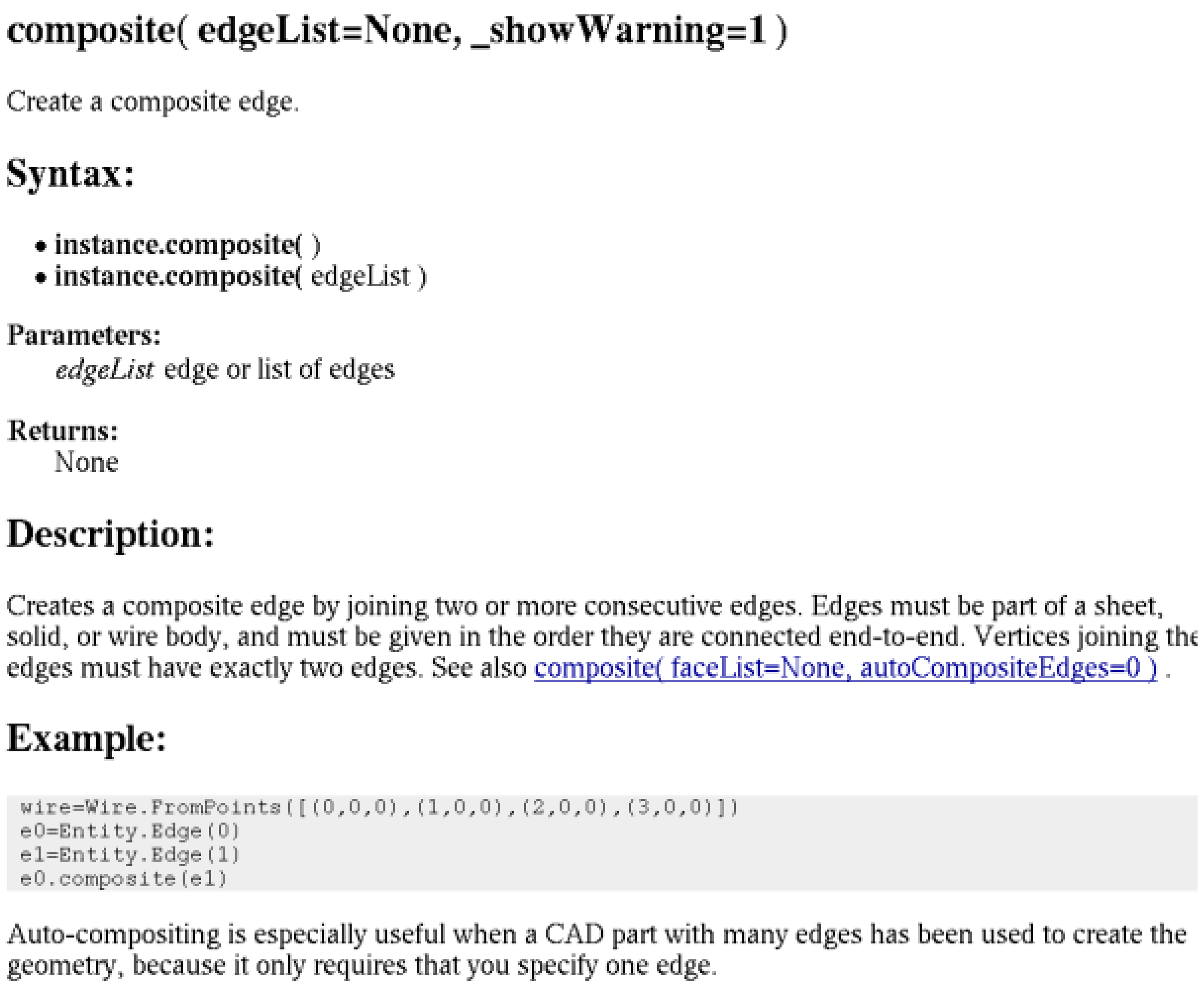
Fig. 1. Sample source code, with annotations

The above source code produces the following output, in brief and extended form. The command *help* generates brief, two line help, while *man* generates the full documentation. In both cases the output is piped to a pager, allowing for scrolling, searching, and other operations. Note that the *\ref* used above does not appear in the output. In general, when the text processor encounters a Doxygen tag it does not understand, it removes it from the output.

```
Draco>>> help(Edge.Edge.composite)
DRACO Documentation for: method composite in module Edge

composite( edgeList=None, _showWarning=1 ) unbound Edge.Edge method
    Create a composite edge.
Draco>>> man(Edge.Edge.composite)
Help on method composite in module Edge:

composite( edgeList=None,  _showWarning=1 ) unbound Edge.Edge method
   Create a composite edge.

   Syntax:
       instance.composite( )
       instance.composite( edgeList )

   Argument(s):
       edgeList - edge or list of edges

   Returns:
       None

   Description:
   Creates a composite edge by joining two or more consecutive edges.
   Edges must be part of a sheet, solid, or wire body, and must be
   given in the order they are connected end-to-end. Vertices joining
   the edges must have exactly two edges. See also
   Face.Face.composite.

   Example:

   wire=Wire.FromPoints([(0,0,0),(1,0,0),(2,0,0),(3,0,0)])
   e0=Entity.Edge(0)
   e1=Entity.Edge(1)
   e0.composite(e1)

   Auto-compositing is especially useful when a CAD part with
   many edges has been used to create the geometry, because it
   only requires that you specify one edge.
```

Fig. 2. Text output for a function

When passed off to Doxygen, the same function's output in HTML form is mostly the same, but contains slightly more advanced formatting, and the hyperlinks that was missing from the text documentation.



composite(edgeList=None, _showWarning=1)

Create a composite edge.

Syntax:

- **instance.composite(**)
- **instance.composite(** edgeList)

Parameters:
edgeList edge or list of edges

Returns:
None

Description:

Creates a composite edge by joining two or more consecutive edges. Edges must be part of a sheet, solid, or wire body, and must be given in the order they are connected end-to-end. Vertices joining the edges must have exactly two edges. See also composite(faceList=None, autoCompositeEdges=0) .

Example:

```
wire=Wire.FromPoints([(0,0,0),(1,0,0),(2,0,0),(3,0,0)])
e0=Entity.Edge(0)
e1=Entity.Edge(1)
e0.composite(e1)
```

Auto-compositing is especially useful when a CAD part with many edges has been used to create the geometry, because it only requires that you specify one edge.

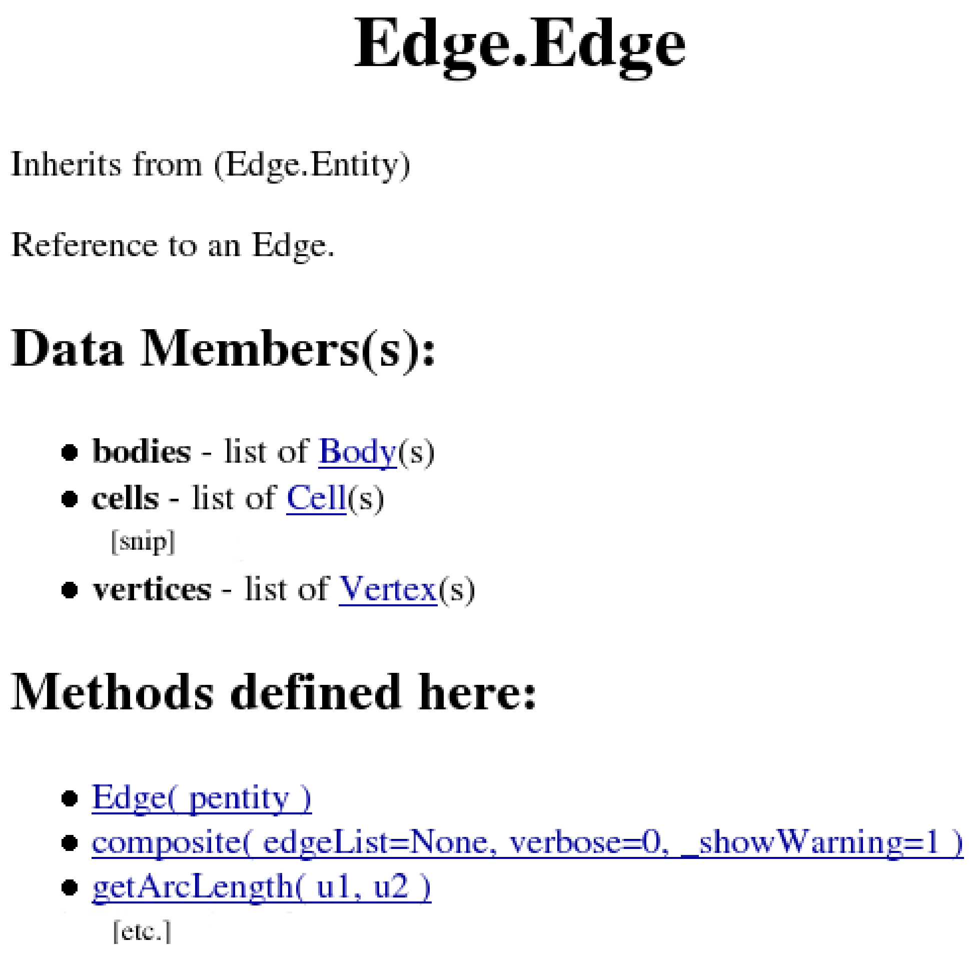
Fig. 3. HTML output for a function.

The brief output for a package, module, or class is a little different, listing the two-line descriptions for all the functions contained therein. Though not shown here, all inherited functions and properties are listed, grouped by the class in which they are implemented.

```
Draco>>> help(Edge.Edge)
DRACO Documentation for: class Edge in module Edge

class Edge(Entity.Entity)
    Reference to an Edge.


    Data Member(s):

    bodies - list of Body(s)

    cells - list of Cell(s)

  [snip]

    vertices - list of Vertex(s)

    -----------------------------------------------------------------
    Methods defined here:

    Edge( pentity )
        Constructor should not be called directly.

    composite( edgeList=None,  verbose=0,  _showWarning=1 )
        Create a composite edge.

    getArcLength( u1,  u2 )
        Return arc length of edge between two parameter positions.
[etc.]
```

Fig 4. Brief text output for a class

The corresponding HTML output is similar, with the addition of automatically generated hyperlinks from the list of functions to the full documentation for those functions. Doxygen also automatically produces PostScript and PDF output, with the same features.



Edge.Edge

Inherits from (Edge.Entity)

Reference to an Edge.

Data Members(s):

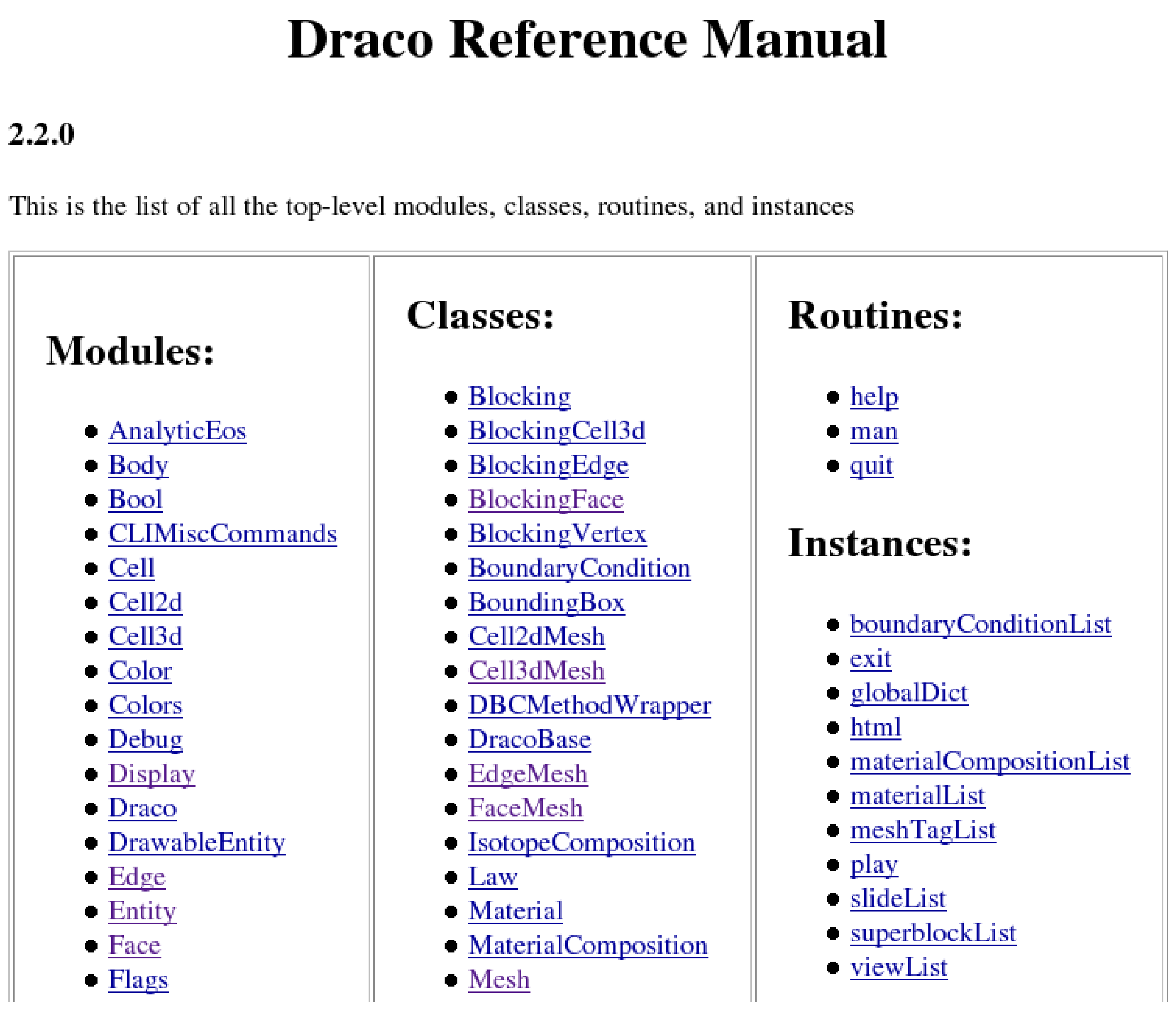
- **bodies** - list of Body(s)
- **cells** - list of Cell(s)
  [snip]
- **vertices** - list of Vertex(s)

Methods defined here:

- Edge(pentity)
- composite(edgeList=None, verbose=0, _showWarning=1)
- getArcLength(u1, u2)
  [etc.]

Fig. 5. HTML output for a class

Another important requirement was that it automatically produce documentation mirroring the structure of DRACO. Like many Python programs, DRACO makes use of import statements and other techniques that cause it to look differently than might be expected based on a simple parse of the source files. For example, the class *Display_* is a singleton instantiated as *Display*. Having it shown under its original class name would have been confusing. Therefore PyHelp always uses the name an object has from the user's perspective, regardless of what its original name or location is. The drawback is that in order to determine the name, the object must be accessible at startup from the command prompt. Objects that are only created as the result of user function calls do not get documented. In general, we work around this requirement by explicitly importing all objects we want to document either at the top level, or as static members of the class or module that produces them. For instance, a *Cell3D* can be queried for its mesh, an instance of *Cell3DMesh*. No instance or definition of *Cell3DMesh* is available without calling a function on *Cell3D*. So, we import *Cell3DMesh* to the top level, where it is documented. All the top-level objects are shown in the HTML index below.



Draco Reference Manual

2.2.0

This is the list of all the top-level modules, classes, routines, and instances

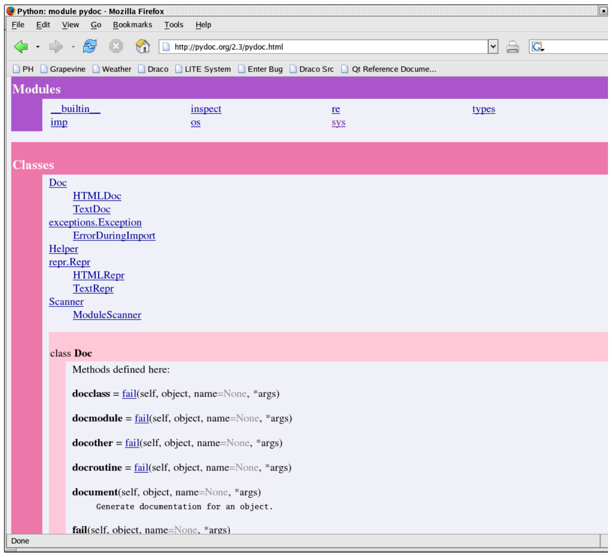
Modules:	Classes:	Routines:
AnalyticEos	Blocking	help
Body	BlockingCell3d	man
Bool	BlockingEdge	quit
CLIMiscCommands	BlockingFace	**Instances:**
Cell	BlockingVertex	boundaryConditionList
Cell2d	BoundaryCondition	exit
Cell3d	BoundingBox	globalDict
Color	Cell2dMesh	html
Colors	Cell3dMesh	materialCompositionList
Debug	DBCMethodWrapper	materialList
Display	DracoBase	meshTagList
Draco	EdgeMesh	play
DrawableEntity	FaceMesh	slideList
Edge	IsotopeComposition	superblockList
Entity	Law	viewList
Face	Material	
Flags	MaterialComposition	
	Mesh	

Fig 6. HTML index

Research and Development

The primary requirements for the DRACO documentation system were ease of writing and updating documentation, ease of navigation for the user, and the ability to generate online and printed documentation. Ideally, we would have used one of the many open-source documentation tools. Doxygen itself would have been an excellent choice, but it supports only C/C++. *Pydoc* produces online and HTML documentation, but doesn't support hyperlinking between functions, any type of formatting, and is profoundly ugly (see Fig. 7)

No existing tool offered the combination of abilities we were looking for. However, *pydoc* came the closest, since it performed sophisticated introspection of Python data structures and could produce online documentation. As an open source application, the Python source code for *pydoc* was available to modify.


Python: module pydoc - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://pydoc.org/2.3/pydoc.html
PH Grapevine Weather Draco LITE System Enter Bug Draco Src Qt Reference Docume...
Done

```
Modules
    __builtin__        inspect        re        types
    imp                os             sys

Classes
    Doc
        HTMLDoc
        TextDoc
    exceptions.Exception
        ErrorDuringImport
    Helper
    repr.Repr
        HTMLRepr
        TextRepr
    Scanner
        ModuleScanner

    class Doc
        Methods defined here:

        docclass = fail(self, object, name=None, *args)

        docmodule = fail(self, object, name=None, *args)

        docother = fail(self, object, name=None, *args)

        docroutine = fail(self, object, name=None, *args)

        document(self, object, name=None, *args)
            Generate documentation for an object.

        fail(self, object, name=None, *args)
```

Fig. 7. Screenshot of *pydoc*'s HTML output

Design

Pydoc consists of a base class, *Doc*, and two output-specific subclasses *HTMLDoc* and *TextDoc*, for HTML and command-line output, respectively. I added a class for Doxygen output, *DoxyDoc*. (See Fig. 8) Doc is responsible primarily for introspection; that is, traversing the hierarchical data structures to locate objects and extract their documentation. Python's hierarchy starts with packages, which contain modules, which contain classes, which contain functions. What is done with the docstring of any given object in the hierarchy is up to the three output classes.

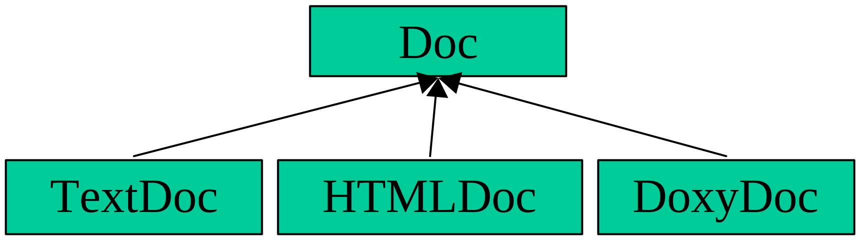


Fig. 8. *PyHelp* class structure.

Implementation and Maintenance

Adapting *pydoc* was straightforward. To produce the user-oriented naming scheme for objects required adding a couple of arguments to each of the recursive function calls, as well as determining the object name at each level. Perhaps the most difficult task was correctly parsing the Doxygen tags. Because the tag syntax is so simple, I used regular expressions in Python. In retrospect, it might have been simpler to write a formal grammar and use a real lexer and parser, as handling all the cases that came up resulted in expressions such as

```
(?:%s(?:[ \t]+|[ \t]*\n[ \t]*\n|\n))((?:.|\n)+?)(?:\n??[\s]*%s).
```

There are also many cases to handle since a function can reside in a module, class, or at the top level, and might be treated differently, but again there are not so many that it is impossible. One goal that I had originally was to employ a stylesheet-like mechanism to use the same logic for both text and Doxygen output. This would in theory eliminate half the output-specific code and facilitate consistency between the kinds of output. However, the complexity of doing all the string manipulation with a stylesheet did not seem worth the benefit.

The overall complexity is not that high, and the only ongoing maintenance is keeping up with new versions of Python that have introduced new-style classes and property objects. The behavior of the inspect module, which does much of the heavy-duty lifting for introspection, changes and typically breaks PyHelp across major releases.

Testing - an unexpected benefit

Having a defined marking for examples also allowed us to perform basic unit testing with no additional work. Benjamin Grover, also of the PMESH team, wrote a script to extract all the source code contained in the example tags, and then execute the code. If the example causes an exception to be raised, the failure is noted in the test log. Executing the example code then provides a benefit to the documentation, because it ensures that all the examples are executable. This was a problem before when developers would change an interface but not update the description or example, causing the example to be out of date. Executing all example code is not foolproof, since it does not confirm the correct behavior, but it is useful.

Conclusion

Modifying *pydoc* to accept Doxygen syntax was a success. PyHelp now provides the DRACO project with high quality, easy to update documentation, as well as limited unit testing. It produces online, HTML, and PDF reference manuals that automatically reflect the structure of DRACO's Python modules and classes from the user's perspective. The end result is a reference manual that is nearly 1500 pages and would have been almost impossible to keep current if produced manually.

Acknowledgements

John P. Clark of Northern Arizona University developed the first HTML index for PyHelp.

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.